

Reach Engine Workflow

For Programmers

Levels Beyond

2710 Walnut Street | Denver, CO 80205 | (303) 495-2424

www.levelsbeyond.com



What Is Reach Engine Workflow?


Simply put, Reach Engine Workflow is an xml programming language for the Reach Engine Studio Automation Engine. These automations allows you to control  almost all actions that will happen with your assets allows you to build automations around your assets. Automations can be as simple as controlling ingest locations and transcodes, setting metadata, or exporting. This course is meant as a guide to help you on your way with programming for Reach Engine. As a note, this course is written in a way to help programmers understand Reach Engine Workflow, if you are not familiar with branching, looping, variables, etc, you may want to start with an intro to programming course before stepping into workflow.

Table of Contents

What Is Reach Engine Workflow?	2
Workflow Basics:	5
What is a workflow Step:	5
Tools	6
Reach Engine Virtual Machine	6
Oxygen XML Editor	6
Metabrowser	6
Documentation	6
Workflow Structure	7
Workflow Branching and Execution	11
Workflow Tip:	11
Subflows	12
Variables	12
Variable Types	12
Setting Variables	13
Programming Structures	16
Looping	17
Environment Reference:	18
Interfacing with external resources:	18
Pulling properties from the localProperties file	18
Path Mapping / localContext	18
Assets In RE Studio	19
find	19
Metadata	21
Save Data Object Step	21
Nimbus:setMetadataValue(s)Step	22
Reaching Outside of RE Studio	23
submitHttpStep	23
runCommandStep	23
Functions RE and Java	25
RE Function Reference	25
Java Class Functions	25
Debugging	26
Execution labels	26
Noop Step	26
Test Step	26
Error Handling:	27
Stalls	27

Fails	27
Bypassing Stalls and Fails	27
Handling an Error	27
Keeping the Workflow From Failing	27
Allowing the parent workflow to continue when a subflow has an issue	28
Workflow Best Practices:	29
Description	29
Commenting.....	29
Step & Transition Naming.....	29
Versioning	29
log statements - execution labels	29
'end' step no-op	29
Advanced Topics.....	30
Coding Outside of Workflow:.....	31
Overview	31
Best Practices	31
When to step out of workflow code:	31
Stay away from:.....	31
Debugging Tips:.....	31
log4j references:	31
Outside Languages	32
Groovy	32
Other (Run Command)	32
Workflow Creation:	33
Test Step — Hello World.....	33
Ingest Directory.....	33
Ingest Directory part 2	33
Set (Select) Metadata on existing Asset.....	33
Read a JSON file into a variable	33
Combine Read & Set	33
Export Timeline with user specified transcode setting	33
Advanced — Query for the asset based on some property	33
Advanced — Set a piece of metadata through API	33

Workflow Basics:

All workflows currently ingested to your system are located under the admin tab -> workflows in Reach Engine Studio. They can be searched, imported, exported, and properties can be assigned all from the UI. There is also a secondary way to get workflows ingested to Reach Engine, utilizing the workflow hotfolder. The location of this hot folder is defined in the `local.reach-engine.properties` file.

Executed (ing) workflows can be observed in the "Workflow" tab in the Reach Engine Studio UI. Workflows can be identified here as running (executing), stopped (paused), failed (user stopped), stalled (system error / workflow error). You can also drill down into the workflow from the UI to watch the steps, child flows, and view the execution labels, get limited log information on stalls, etc...

Running workflows: To run a workflow, you can call it from the API, from a hotfolder, from a CRON, or from the UI. From the main Content screen in Studio or Access open the Actions menu. Opening the menu with no assets selected will show you the available workflows that do not require a subject to start. Now select an asset, workflows for that asset type, as well as workflows that do not require a subject are now show. Select multiple types, and you will see workflows only available for Both the asset types selected (note — some of these are actions which fire multiple workflows, rather than singular workflows). Later we will get into how to trigger workflow execution from hotfolder, cron, and api.

If your log levels are properly set, you will see your workflow running in the reach-engine log file, here you can see your log statements (test step).

What is a workflow Step:

Simply put, the xml workflow step references a custom written piece of Java code inside Reach Engine Studio.

Tools

First let's start with programming Environment, and the tools necessary for us to work with Reach Engine on Workflow.

Reach Engine Virtual Machine

The VM is the first and most important tool we are going to work with, it gives you a playground environment separate from your primary Test and Production servers to do your development against. Lets first start by getting this system up and running and getting UI, Mounts, and ssh access to your VM.

Oxygen XML Editor

There are a lot of XML editors out there, but this one seems to fit the bill best for our internal development, hence it's recommended as your primary tool. Benefits like auto completion and code validation, as well as the XPath builder make this one top of our list.

(Download Oxygen get it registered, open a workflow, show auto-completion)

Metabrowser

This is an internally developed and maintained (uhh, yeah, barely) tool that makes it easy to see the data structures you will be looking at in RE, as well as help to generate queries when working with the queryStep in workflow. Note at this writing it's buggy, but it's a staple to use, so if it seems to be running funky, close and re-open it, that cures 99% of its ailments.

(Open, connect to the VM, show the UI, show properties, associations, beware of calculated.....)

(Show location of query step creation, how to create a query)

Documentation

As of this writing there are a few good docs to keep handy when working on workflow. Workflow the manual will give you a pretty large overview on what workflow is, and walk you through some basic examples, but when we are working, it's not a reference I'm always grabbing for. These are what we heavily rely on when building workflows:

Workflow Step Reference

Workflow Function Guide

Workflow Examples (Reference Flows)

Workflow Structure

Next let's talk a bit about the structure of RE Workflow. RE Workflow as I mentioned before is an XML based programming language. Here is a simple example:

```
<workflow xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://levelsbeyond.com/schema/workflow"
  xmlns:nimbus="http://levelsbeyond.com/schema/workflow/nimbus"
  xsi:schemaLocation="http://levelsbeyond.com/schema/workflow http://www.levelsbeyond.com/schema/latest/studio.xsd"

  id="sendTimelineToFtp"
  name="Send Timeline to FTP"
  description=""
  subjectDOClassName="Timeline"
  sdkVersion="4.2">

  <initialStepName>uploadFtpStep</initialStepName>

  <nimbus:ftpFileStep
    name="uploadFtpStep"
    sourceFileExpression="${subject.mainMezzanineContent.file}"
    ftpServerExpression="${server}"
    ftpUserExpression="${username}"
    ftpPasswordExpression="${password}"
    ftpFolderExpression="${folder}">

    <transition condition="${true}">
      <targetStepName>end</targetStepName>
    </transition>
  </nimbus:ftpFileStep>

  <noopStep name="end" executionLabelExpression="FTP workflow complete"/>

  <contextDataDef dataType="String" name="server" label="FTP server address" userInput="true" required="true"></contextDataDef>
  <contextDataDef dataType="String" name="username" label="FTP username" userInput="true" required="true"></contextDataDef>
  <contextDataDef dataType="String" name="password" label="FTP password" userInput="true"></contextDataDef>
  <contextDataDef dataType="String" name="folder" label="FTP folder" userInput="true"></contextDataDef>

</workflow>
```

In the open workflow tag, we have definitions for namespace nimbus (one of our internal name spaces), the schema location (this is what Oxygen uses for your validation and code completion), then some initial properties.

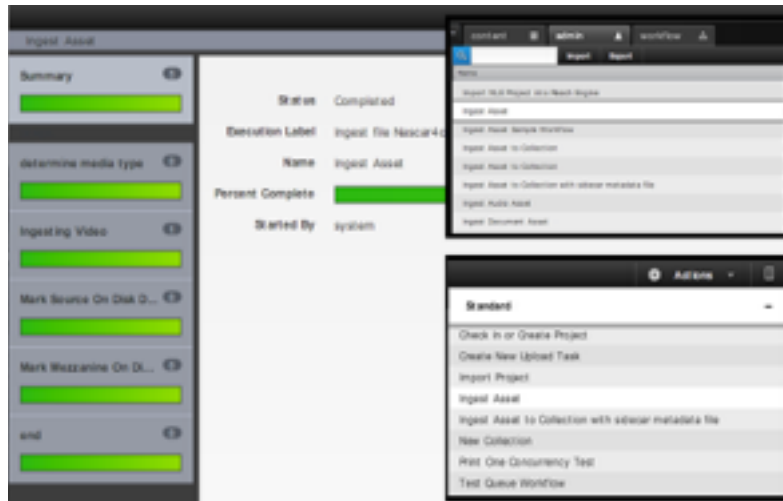
Id:

Id is our unique identifier, **anytime you want a new workflow in the system this is what needs to be unique.**

Versioning the workflow, you leave it the same. This is also the ID you use when calling this workflow from the API, from a Hotfolder, or if you want to use this as a subflow.

Name:

This is the name of the workflow in the admin workflow tab in RE Studio, as well as what this will show up as in the actions menu. Basically **what you and the users are going to see** when they go to run this workflow, this does not have to be unique, but in most cases unique is recommended.



Description:

(Best Practices) This is a good place to put a description of what this flow does, so in 6 months from now when you go look at it you remember what the heck it was for. A good rule of thumb is a quick description followed by what goes in and what comes out. Lets fill in the one above:

```
<workflow xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://levelsbeyond.com/schema/workflow"
  xmlns:nimbus="http://levelsbeyond.com/schema/workflow/nimbus"
  xsi:schemaLocation="http://levelsbeyond.com/schema/workflow http://www.levelsbeyond.com/schema/latest/studio.xsd"

  id="sendTimelineToFtp"
  name="Send Timeline to FTP"
  description="Workflow to send a Timeline main mezzanine to an ftp server.
Subject: Timeline
Takes:
  (String required) server: ip address of ftp server
  (String required) username: ftp username
  (String optional) password: ftp password
  (String optional) folder: ftp folder to place the file in
Returns: Nothing"
  subjectDOClassName="Timeline"
  sdkVersion="4.2">
```

...

SubjectDOClassName:

Subject Data Object Class Name: This is what you have to have selected in RE to run this flow, and what get's passed to the flow as the "subject". As you can see, this workflow references "Timeline" (which is a Video Object in RE Land)

Subjects:

- Timeline (Video)
- AssetMaster
- ImageAssetMaster
- AudioAssetMaster
- DocumentAssetMaster
- Collection
- Project
- Clip

```
<workflow xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://levelsbeyond.com/schema/workflow"
  xmlns:nimbus="http://levelsbeyond.com/schema/workflow/nimbus"
  xsi:schemaLocation="http://levelsbeyond.com/schema/workflow http://www.levelsbeyond.com/schema/latest/studio.xsd"

  id="sendTimelineToFtp"
  name="Send Timeline to FTP"
  description="Workflow to send a Timeline main mezzanine to an ftp server.
    Subject: Timeline
    Takes:
      (String required) server: ip address of ftp server
      (String required) username: ftp username
      (String optional) password: ftp password
      (String optional) folder: ftp folder to place the file in
    Returns: Nothing"
  subjectDOClassName="Timeline"
  sdkVersion="4.2">
```

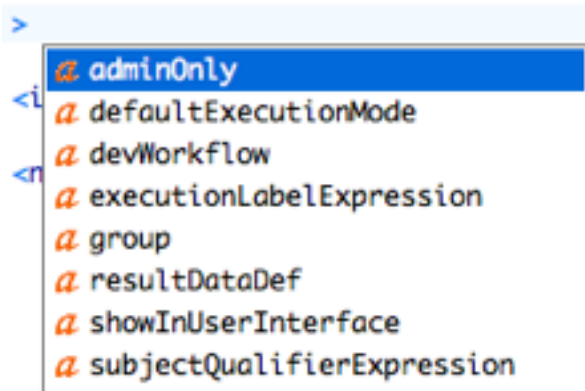
Referencing **Timeline** means that **this workflow will only be available in the UI if a video is selected**, it also **must be passed a Timeline object if it is to be used in a subflow**. You will also notice that a special object is used in this workflow called "subject". This is what the user has selected in the UI, or passed into this workflow when this workflow is run as a subflow, since you know you have restricted it to Timeline, you know that is what the object is.

```
<nimbus:ftpFileStep
  name="uploadFtpStep"
  sourceFileExpression="${subject.mainMezzanineContent.file}"
  ftpServerExpression="${server}"
  ftpUserExpression="${username}"
  ftpPasswordExpression="${password}"
  ftpFolderExpression="${folder}">
```

This is a good segway into what properties are available on an object in workflow. For this you use Metabrowser, or reference flows (examples). Lets take a look at what properties and associations are on Timeline. Lets do that now.

More Control Over Your Workflows:

If you open Oxygen with this workflow (in your references, you can put your cursor prior to the closing caret and use command space, you will be prompted with other things you can insert in the workflow header:



We can build more restrictions into the workflow by using these flags. The auto tool tip will give you most of the info you will want to know about these options, feel free to ask if you have questions about what these options may mean.

adminOnly sets the admin flag for a workflow, so only admin and system have access to this workflow.

defaultExecutionMode sets the execution mode of this workflow

devWorkflow sets this as an internal hidden flow, which cannot be selected in the UI, and can only be viewed executing while the internal workflows flag is selected. (Good way to hide cron flows from the user)

executionLabelExpression sets the execution label seen in the UI (use this!!)

group sets the default group for this workflow in the UI

resultDataDef the return variable for this workflow

showInUserInterface sets the default show in the UI, (do you want the users to see this workflow in the Actions tab?)

subjectQualifierExpression an expression used to control the visibility of this workflow based on parameters (such as specific metadata settings on this asset)

engineAppVisibility used to control visibility of this workflow to specific engine applications, defined by the user or by Levels Beyond (Premier/Prelude/Access)

hasDownload used to flag access (through API calls) that the result of this workflow is a downloadable file

Workflow Branching and Execution

Before we go much farther, let's talk about the way Workflow Executes. For those of you old enough to remember Basic A with the "Go To" statements, this will probably bring up some flashbacks.

Reach Engine works in Steps. Let's just concentrate on a couple things in the steps that control where things are going:

Name:

The Name is important, **it is the unique identifier** of this step, used throughout this workflow.

Transition:

This is the decision tree of where this workflow will go next, with the condition that will be evaluated and the next step name to go to. Something important to note, these are evaluated like an if – else if, the first one that evaluates to true is where this workflow is going to go next. An Important note, you will always need a catch all statement at the end, think of it as the "else" at the end of if : else if : else syntax.

InitialStepName:

This is where the workflow will **start**.

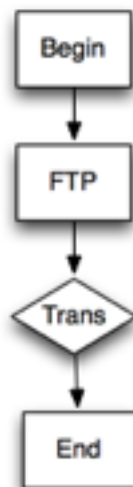
NoOpStep: end

This is best practice recommended, so you can always see that your workflow went to an end step.

Let's follow the path of this workflow.

Workflow Tip:

If you are one of the devs that likes to diagram everything out, then visualize the step like this:



Subflows

Reach Engine has the ability for a workflow to call another workflow through one of its steps:

We refer to this workflow as a child workflow. This makes for a cool way to implement code re-usability, or if you prefer, a way to implement functions or methods. A great example is the FTP flow. This flow can be repurposed as a primary flow, for the users to always select a file and enter in information, or as a child, to be called by automation. We can also use the `executeSubflowStep` to perform parallel processing from an array data structure, this is a very powerful tool to have under your tool belt, and with all major power wielding tool comes the disclaimer: "With Great Power Comes Great Responsibility" we will get into parallel processing from subflows a bit later.

```
<executeSubflowStep
  name="run another workflow"
  targetWorkflowId="childWorkflowID"
>
</executeSubflowStep>
```

Variables

Variables are pretty important to programming languages, so let's look at them next. In the RE Workflow world, Variables are called "**Context Data Definitions**" and, **they are all defined at the bottom of the document**. It takes a bit of getting used to, but that's the way the XML validator likes it.

Variable Types

These variable types are available in RE Workflow:

Types:

- Boolean
- Date
- Date/Time
- Directory
- Double
- Email
- File
- Integer
- JSON
- Name Value Pair
- String
- XML
- Data Object

Most of these types are self explanatory, but lets talk about a couple of them that may not be.

(NVP, DO, Date vs Date/Time)

Date/Time is only in workflow — you cannot store time references in a date object in Studio.

If you need time references, store your dateTimes as STRINGS.

Setting Variables

Next lets look at the number of ways to set them

Hardcoding / Default Values:

I wont start some philosophical debate on why you would want to hardcode a variable, let's just look at how we make it happen. Pop open Oxygen again, and place your cursor after all the definitions is the variable "folder", add a space and the autocomplete should give you your list of options. Note the one called "defaultDataExpression" that's the one you want if you wanna hardcode something into a variable of a workflow. This is also how you set defaults, you can use expressions in here also for things you know will be available at the runtime of this workflow.



Dynamically Setting:

Reach Engine Workflow has a step specifically to set a variable, the setContextDataStep, lets try it, open Oxygen and we are going to make a new variable and set the value.

```
<contextDataDef name="foo" dataType="String"></contextDataDef>
```

We will do this prior to running the ftp upload. If you add a couple of carriage returns and add an open caret, then start typing set the step should pop up, then select it and hit enter. I like to reformat this a little. Lets set a new variable we will define called "Foo" to the timeline name and the ftp point so we can print out a debug statement for us later.

```
<setContextData
  name="set foo"
  targetDataDef="foo"
  valueExpression="{subject.name} is uploading to {server}" >

  <transition condition="true">
    <targetStepName>uploadFtpStep</targetStepName>
  </transition>
</setContextData>
```

Notice I am calling subject to get the name, that I've added a catch all transition, and that I've wired where it will go. I also need to define the variable at the bottom of the flow.

Cool. Now if you point the start to the new step, we will set this variable during the workflow.

Passed from Parent:

Let's add a little onto executing subflows. There is obviously the need to pass variables from the parent to child when calling another workflow. There are a couple things to know when doing this.

Passing the Subject:

pretty easy, not really anything surprising here.

```
<executeSubflowStep
  name="run another workflow"
  targetWorkflowId="childWorkflowID"
  subjectChangePath="subject"
>

</executeSubflowStep>
```

Passing a variable:

Passing a variable from parent to child is also not too difficult, you can assign the parent variable to any child variable also, here we are assigning "foo" from the parent to "bar" in the child.

```
<executeSubflowStep
  name="run another workflow"
  targetWorkflowId="childWorkflowID"
  subjectChangePath="subject"
>

  <subflowContextDataMapping parentDataDef="foo" subflowDataDef="bar"/>
</executeSubflowStep>
```

Passing the Subject to a Variable:

Here I am assigning the subject (a Timeline in this workflow we are using as an example) to the child variable "foo". Better make sure the variable types match – this is a Data Object.

```
<executeSubflowStep
  name="run another workflow"
  targetWorkflowId="childWorkflowID"
  subjectChangePath="subject"
  subflowTargetDataDef="foo"
>

  <subflowContextDataMapping parentDataDef="foo" subflowDataDef="bar"/>
</executeSubflowStep>
```

Result Data Def:

This one is uber useful. If you run a child workflow and have it return a variable, run a run command, use groovy, etc..... you can set a variable as a result variable from a step:

Once this step returns, it will overwrite the "foo" variable with its result. Better make sure that the types match.

```
<executeSubflowStep
  name="run another workflow"
  targetWorkflowId="childWorkflowID"
  subjectChangePath="subject"
  subflowTargetDataDef="foo"
  resultDataDef="foo"
>

  <subflowContextDataMapping parentDataDef="foo" subflowDataDef="bar"/>
</executeSubflowStep>
```

Hotfolder:

Setting a variable from a hotfolder is pretty easy, you simply add the definition you would like to set to your hotfolder config (note this requires a restart to apply) and voila. Everything on the right side of the equals sign will be stored in the variable.

```
workflow.hotfolder.hotfolderID.foo=value for foo
```

Programming Structures

We've already hit on the if – else if – else structure, but to make sure you get it, let's look at something crazy:

this step is there just as an if – else if – else, validating something for each transition. The transitions are evaluated top to bottom, with **the first one that evals to true evaluated**. To be clear once one of them is true, no more transitions are evaluated.

```
<noopStep name="validate exif">
  <transition condition="{exifJobID == null}">
    <targetStepName>find collection</targetStepName>
  </transition>
  <transition condition="{assetVersion.metadata.exifDescription == null}">
    <targetStepName>find collection</targetStepName>
  </transition>
  <transition condition="{assetVersion.metadata.exifCaptionWriter == null}">
    <targetStepName>find collection</targetStepName>
  </transition>
  <transition condition="{assetVersion.metadata.exifHeadline == null}">
    <targetStepName>find collection</targetStepName>
  </transition>
  <transition condition="{assetVersion.metadata.exifKeywords == null}">
    <targetStepName>find collection</targetStepName>
  </transition>
  <transition condition="{assetVersion.metadata.exifByLine == null}">
    <targetStepName>find collection</targetStepName>
  </transition>
  <transition condition="{assetVersion.metadata.exifByLineTitle == null}">
    <targetStepName>find collection</targetStepName>
  </transition>
  <transition condition="{assetVersion.metadata.exifCopyrightNotice == null}">
    <targetStepName>find collection</targetStepName>
  </transition>
  <transition condition="{assetVersion.metadata.exifCredit == null}">
    <targetStepName>find collection</targetStepName>
  </transition>
  <transition condition="{assetVersion.metadata.exifSource == null}">
    <targetStepName>find collection</targetStepName>
  </transition>
  <transition condition="{assetVersion.metadata.exifDateTimeOriginal == null}">
    <targetStepName>find collection</targetStepName>
  </transition>
  <transition condition="{assetVersion.metadata.exifCity == null}">
    <targetStepName>find collection</targetStepName>
  </transition>
  <transition condition="{assetVersion.metadata.exifLocation == null}">
    <targetStepName>find collection</targetStepName>
  </transition>
  <transition condition="{assetVersion.metadata.exifCategory == null}">
    <targetStepName>find collection</targetStepName>
  </transition>
  <transition condition="true">
    <targetStepName>find good collection</targetStepName>
  </transition>
</noopStep>
```


Looping

Looping is possible in workflow, but it takes some work. We use this pretty often when trying to chunk up some of the parallel processing into manageable chunks (yep, you gotta worry about that right now in writing workflows – remember the “great power, great responsibility” statement?)

Let's break down the portions of a for loop.

Loop counter; Loop Code Contents; Increment the counter; Exit Condition

You may or may not catch where this is going..... each one of these is probably going to be another step in workflow. Let's do something simple,

for(i = 0; i<5; i++) {show i;}

The set and increment are pretty straight forward, as is printing the iteration. Note how the transitions work to either send you into another loop iteration or let you out.

```
<setContextData
  name="set counter"
  targetDataDef="loopCounter"
  valueExpression="0" >

  <transition condition="true">
    <targetStepName> print loop counter</targetStepName>
  </transition>
</setContextData>

<noopStep
  name="print loop counter"
  executionLabelExpression="print iteration ${loopCounter}"

  <transition condition="true">
    <targetStepName>increment loopCounter</targetStepName>
  </transition>
</noopStep>

<setContextData
  name="increment loopCounter"
  targetDataDef="loopCounter"
  valueExpression="${loopCounter + 1}" >

  <transition condition="${loopCounter < 6}">
    <targetStepName>print loop counter</targetStepName>
  </transition>
  <transition condition="true">
    <targetStepName>end</targetStepName>
  </transition>
</setContextData>

<noopStep name="end">
```

Environment Reference:

Interfacing with external resources:

Reach Engine has the concept of a virtual filesystem. This allows us to switch the low level filesystem without having to update the database with new hard links to all our internal resources. for an example: a file at:

```
/reachengine/media/mezzanines/path/to/file.mov
```

is referenced in reach engine as: mezzanines/path/to/file.mov, and there is a resource (the local properties file) that explains that mezzanines is really /reachengine/media/mezzanines.

because of this reason, there may be times you see an error in studio while trying to reach a file that is in a non-defined area. Park this as a reminder in the back of your mind if you see an error that states you are accessing a file that is not referenced by a filesystem root, you probably need to update your local properties file.

As a side note — as long as the file you are trying to reach is represented by a filesystem root, you are good to reference it with the root name virtual path (eg: mezzanines/path/to/file.mov) and the direct reference to the location to the file (/reachengine/media/mezzanines/path/to/file.mov)

Pulling properties from the localProperties file

There will come a time when you have a global variable to your system that you would like to abstract from the workflow. An ftp address, a user name/password for an external server, something that may need to be referenced by multiple flows. This is how you do it:

```
#sysconfig('propertyName')
```

This will utilize the Reach Engine function sysconfig() to evaluate the property from the local properties file and toss it in place of this function.

Path Mapping / localContext

Reach Engine is built on a flexible platform that will allow you to reach out to multiple separate systems for data, independent of operating system utilizing resources like the exec server, but that leaves you with a problem, how to correctly represent the asset being passed around. Windows will have a different file location vs linux vs mac. We have built the path mapping service into Reach Engine to handle just that. In most cases where a supported integration has been written, this is handled behind the scenes based on definitions for the translation in the localContext file. We also offer a function to map a path when needed for workflow.

Assets In RE Studio

find

Ok, I think we've gotten a good intro on some basic things in workflow. Now lets look at how to find a timeline or asset.

```
<queryStep name="check for duplicate asset"
  targetDataObjectClass="AssetVersion"
  resultDataDef="existingAssetVersion">
  <transition condition="{existingAssetVersion != null}">
    <targetStepName>set existing asset master</targetStepName>
  </transition>
  <transition condition="=true">
    <targetStepName>set skip mezzanine flag</targetStepName>
  </transition>
  <criteria>
    <![CDATA[
    <criteria>
    <and>
      <condition property="name" op="eq"><test value="{fileToIngest.name}"></condition>
      <condition property="currentVersionFlag" op="eq"><test value="true"></condition>
    </and>
    </criteria>
    ]]>
  </criteria>
</queryStep>
```

This is a pretty typical QueryStep. This is what we use in workflow to query the databases (yep, more than 1 so you gotta use the step). This step is taken from the default ingest, it looks for an assetVersion (this is the lowest level looking directly for the files) that is the current version for an object, and has a name that matches a file name. This is going to return an Array List of data objects matching the type of your targetDataObject. Then you can use Metabrowser to find out where you can traverse in terms of associations and attributes available from this Data Object.

Most common finds are going to be on AssetMaster (for any of Images/Documents/Audio/Other), specific classes for one of those object types (eg: ImageAssetMaster), Timeline (Video Object), AssetCollection, and Project. There could be reasons you go to assetVersion (like if you are looking for something archived) but most of the time you will stick in these objects. You can see a full list of Data Objects in Metabrowser.

A quick example on finding all images with specific metadata:

Let's break this one down, there is a lot of example in this query. The find class is ImageAssetMaster, so we

```
<queryStep
  name="find transition assets"
  targetDataObjectClass="ImageAssetMaster"
  resultDataDef="transitionAssets"
>

  <transition condition="true">
    <targetStepName>print found assets</targetStepName>
  </transition>

  <criteria>
    <![CDATA[
      <criteria>
        <and>
          <condition property="metadata.categories.name" op="notin" nullTargets="false">
            <test value="Approved"/>
          </condition>
          <condition property="metadata.categories.name" op="in" nullTargets="false">
            <test value="Review"/>
          </condition>
          <or>
            <condition property="metadata.foo" op="isnull" nullTargets="true">
              <test value=""/>
            </condition>
            <condition property="metadata.foo" op="eq" nullTargets="true">
              <test value=""/>
            </condition>
          </or>
        </and>
      </criteria>
    ]]>
  </criteria>
</queryStep>
```

are looking for Images. The criteria has an and with a nested or, properties outside the or will be anded together. Either of the properties in the or statement will have to be true to satisfy the and statement. So the condition reads: Not in the category Approved and in the category Review and (foo is null or foo is "").

The queries can get quite complex with many many conditions, and for multiple values on a single condition (This would read Not in Approved AND Not in Review.):

```
<condition property="metadata.categories.name" op="notin" nullTargets="false">
  <test value="Approved"/>
  <test value="Review"/>
</condition>
```

Metadata

One of the main reason to have an Asset Management system is Metadata. Lets look at ways to set it on assets in Studio from Workflow.

Save Data Object Step

When looking to set a lot of properties on an asset, this is the primary method.

Here we are just throwing a json data structure at the saveDataObjectStep. This works when the json is structured in a manner where the keys match the metadata ids in RE Studio and the values fall in line with your value definitions. (Don't try to send a Date to an integer field, and **CASE MATTERS**)

```
<saveDataObjectStep name="save timeline metadata"
  dataObjectExpression="{timeline.metadata}"
  nameValuePairsDataDef="metadata"
  jsonValuesDataDef="jsonMetadata"
>
  <transition condition="true">
    <targetStepName>check for asset category</targetStepName>
  </transition>
</saveDataObjectStep>
```

Here we are setting a couple metadata properties using the saveDataObjectStep, directly on the timeline mainMezzanineContent.

```
<!-- timeline archive -->
<saveDataObjectStep name="mark timeline mezz archived"
  executionLabelExpression="Mark asset content ${timeline.id} as archived"
  pctComplete="90"
  dataObjectExpression="{timeline.mainMezzanineContent}"
  devStep="true">
  <transition condition="true">
    <targetStepName>truncate</targetStepName>
  </transition>
  <property name="truncatedFlag">true</property>
  <property name="restoreKey">${targetFileString}</property>
</saveDataObjectStep>
```

Nimbus:setMetadataValue(s)Step

A couple more ways to skin the proverbial cat, nimbus has steps that allow us to save a name:value pair

```
<!-- apply exif metadata to timeline -->
<nimbus:setMetadataValuesStep name="apply metadata to timeline"
  displayName="Apply metadata to the video"
  targetExpression="{ timeline }"
  propertyPrefixExpression="exif"
  metadataPairsExpression="{ exifMetadata }"
  metadataGroupExpression="{ reviewMetadataGroupName }"
  createIfNotFoundExpression="{ true }"
  continueOnException="true"
  pctComplete="70">
  <transition condition="{ true }">
    <targetStepName>set location meta timeline</targetStepName>
  </transition>
</nimbus:setMetadataValuesStep>
```

array (similar to doing it in JSON) this is useful when you want to save AND CREATE FIELDS on a bunch of metadata on an asset.

Also shown is a way to update a single metadata value in a step using the nimbus step.

```
<!-- apply location metadata to timeline -->
<nimbus:setMetadataValueStep name="set location meta timeline"
  propertyNameExpression="originalLocation"
  targetExpression="{ timeline }"
  valueExpression="{ fileToIngestPath }"
>
  <transition condition="{ true }">
    <targetStepName>print exifCaption</targetStepName>
  </transition>
</nimbus:setMetadataValueStep>
```

Reaching Outside of RE Studio

Workflow has a few methods that allow you to reach outside RE studio to pull in metadata or run external scripts.

submitHttpRequest

Here is a simple request to a Sharepoint endpoint that sets a couple variables we can later parse like the HTTP response code and response payload data def.

```
<submitHttpRequest
  name="CallSharepoint"
  urlExpression="${sharepointURL}"
  requestMethodExpression="POST"
  responseCodeDataDef="jsonHttpResponseCode"
  responsePayloadDataDef="sharepointJSONResponse">
<transition condition="=true">
  <targetStepName>End</targetStepName>
</transition>
<requestHeader name="Authorization">Basic passwordRemoved</requestHeader>
<requestHeader name="Accept">application/json</requestHeader>
<requestHeader name="Content-Type">application/json</requestHeader>
<requestPayloadItem name="data"> {'OriginalFilename':'${originalFilename}','Filename':'${filename}','Episode':'$
{episode}','SenderEmail':'${senderEmail}','Revision':'${revision}' }
</requestPayloadItem>
```

runCommandStep

The runCommandStep almost deserves it's own class, it's that versatile. Lets just take the tip of the iceberg to start. Suppose we have an application we would like to run on the filesystem.

The run command step will let us reach out to the filesystem and run that application. Similarly we can run Scripts and grab the output for processing.

```
<runCommandStep name="rename file"
  executablePathExpression="/bin/mv"
  executionLabelExpression="renaming ${result.absolutePath}">
<transition condition="=true">
  <targetStepName>set fileToIngest</targetStepName>
</transition>
<arg>${result.absolutePath}</arg>
<arg>${#filepath(result)}/${directoryMeta.get('tarballDirectory1').asText()}_${result.name}</arg>
</runCommandStep>
```

The ends to this are a bit limitless, let's talk about at one more thing. If we use executableNameExpression instead of executablePathExpression, we can allow the system to reach out to names of functions we have defined for exec servers, thus allowing us to reach out to external systems to run these applications and scripts for us.

```

<runCommandStep name="open command"
  displayName="open ${subject.name}"
  executableNameExpression="open"
  preferRemoteExecution="true"
  remoteHostExpression="${usrIP}"
>

  <arg>-a</arg>
  <arg>${applicationToOpen}</arg>
  <arg>${sourceFilePath}</arg>
</runCommandStep>

```

In this case we are also specifying the host ip address for this to run on, but this is optional. The power of this? Getting the processing off the linux system, or extending the capabilities of RE Studio outside the local Linux Environment. For Example: Mac Caption is not available on Linux, but we can install a remote exec server on Mac and give it the path to the binaries we want to use then use run command to run them.

```

<runCommandStep name="Ruby Copy File"
  executionLabelExpression="Copy file"
  executablePathExpression="${rubyExec}"
  pctComplete="5"
  stdoutDataDef="rubyOut"
  stderrDataDef="rubyError">

  <transition condition="true">
    <targetStepName>end</targetStepName>
  </transition>

  <arg>${rubyInfoFetch}</arg>
  <arg>${myFilepath}</arg>
  <arg>${myFolderOption}</arg>

</runCommandStep>

```


Functions RE and Java

Let's talk functions for a minute. Reach Engine Studio has a bunch of functions built into it to make your day to day programming a better experience. These functions are in the Function Reference Guide that accompanies this document. I'll also demonstrate how to call java class functions in case you need to do so while programming in workflow.

RE Function Reference

Here is an example of how to use the built in Reach Engine java functions to get the base file name of a subject.

The function Reference Guide contains all the functions that can be run in this manner.

```
<setContextData name="Set base file name"
  valueExpression="$ { #baseFilename(fileToIngest) }"
  targetDataDef="filename">
  <transition condition="$ {true}">
    <targetStepName>Go Somewhere</targetStepName>
  </transition>
</setContextData>
```

Java Class Functions

Since Reach Engine Studio is running in a Java environment, you have access to java classes, though this is probably not how you imagined running them.

This reaches out to java.lang.Integer for the function parseInt(). Using the java classes comes up rarely, so this one is a good one to park somewhere in the back of your mind.

```
<setContextData name="try again"
  targetDataDef="tryCount" valueExpression="$ {T(java.lang.Integer).parseInt(tryCount) + 1}">
  <transition condition="$ {tryCount > 3}">
    <targetStepName>copy transcode result</targetStepName>
  </transition>
  <transition condition="$ {true}">
    <targetStepName>export clip</targetStepName>
  </transition>
</setContextData>
```

Debugging

We had to get to it sooner or later. Debugging is pretty simple in workflow. The few things you will want to know:

Execution labels

These are your friend, as well as friend to your users. By setting execution labels you can give yourself data about variables when that step is running. Use them, love them.

```
<noopStep
  name="loop code"
  executionLabelExpression="Hi Mom!! print iteration ${loopCounter}"

  <transition condition="${loopCounter < 5}">
    <targetStepName>increment loopCounter</targetStepName>
  </transition>
  <transition condition="true">
    <targetStepName>end</targetStepName>
  </transition>
</noopStep>
```

Noop Step

Noop means no operation, simply a step for you to use for transitions and execution labels, see the above for an example.

Test Step

The Test Step is a log output step, it will spit some output out to the logs for you

Keep in mind that if the output is going to be long, also include an execution label so Reach doesn't try to spit the entire output to the execution label.

```
<testStep name="log conversion results" outputExpression="${stdout}, ${stderr}" >
  <transition condition="${true}">
    <targetStepName>end</targetStepName>
  </transition>
</testStep>
```

With that We have the basis covered, I'll add an advanced section to here for Groovy. Included you should have received the workflow SDK and Function Reference guide. Also you will be receiving some base workflows for reference. Feel free to reach out to your Levels Beyond Workflow Architect if you have questions. Thank you, and happy coding.

Error Handling:

Stalls Versus Fails and Issues Bypassing Them

Stalls

A stall occurs when a step receives an error and is not set to continue on exception. Stalls should be unexpected errors that mean something truly unexpected has happened and requires investigation. Stalls halt the workflow, as it should.

Fails

A fail occurs when a failWorkflowStep is specifically called in a workflow. This means the workflow is specifically set up to fail under certain circumstances by design. Failing a workflow should be done when a planned problem occurs. If this planned problem needs to create immediate attention or a prompt for investigation, a fail step will stop the workflow and allow users to more clearly identify where or why the problem has occurred.

Bypassing Stalls and Fails

It is important to note that letting the workflow complete when there is an issue instead of letting the workflow stall or fail can be incredibly hard to debug since every step is labeled as completed and the issue could have occurred seemingly anywhere when a user is looking at the steps. If you must allow the workflow to continue running, I recommend adding an emailStep or writing info to a .txt file between the exception handling step and the end step that notifies the user and/or admin of the issue. An email notification or some way to view issues outside of the workflow UI can simplify the debugging process.

Handling an Error

Most steps that will stall will be steps that set a contextDataDef. In this case, it should be possible to set the step property "continueOnException=true" and then for the first transition step check to see if your resulting data def is null. If it is null, it means the step did not complete properly and this transition will allow you to move to wherever you need to go in the workflow to handle the exception properly.

Keeping the Workflow From Failing

If you currently have a failWorkflowStep being called and you no longer want the workflow to fail, you can make this step a noopStep and then transition it to end. The noopStep will still be a step that is visible in the UI to show information about why there was an issue, but the workflow will no longer fail.

Allowing the parent workflow to continue when a subflow has an issue

The best way to handle errors and allow the workflow to continue can get tricky depending on what the subflow is designed to do, but there are 3 general cases.

1. The subflow does not return any values to the parent workflow.

In this case, usually the parent can continue freely because the parent was not dependent on the subflow for any information. If you need to handle an issue in a subflow in this case, you can modify the subflow to return a boolean instead of nothing. Return true if there was no issue or return false if there was an issue.

View the previous section to help see where to set the boolean to true or false. In the parent workflow, you can take this return and transition based on whether the return is true or false.

2. The subflow returns a single value to the parent workflow.

In this case, we can use similar functionality that was described in the previous section. Make sure the subflow returns a null value if there was an issue. Then you can do a transition check in the parent to see if the returned value was null. If the value is null, transition to error handling, otherwise continue as planned.

3. The subflow returns an array of values to the parent workflow.

This is the trickiest case of them all and can be handled in many ways.

The simplest way to handle this would be to take the returned array and loop through each position checking if the position is null. Create a new array that consists of only the non-null positions of the original array. Notes for each of the positions that were null are hopefully handled in the subflow as discussed in the previous section.

If you need the bad array positions to still be in play for whatever reason, you will need to handle the issue down the road for however it suits your case.

Workflow Best Practices:

Description

Use it, write a short sentence to describe the workflow, include the subject type, returns, and variables it takes. When you forget this workflow in a month, it will make a quick short read to remember it, or if you are sharing the responsibility across a team, it will save you a lot of questions on what this workflow is supposed to do.

Commenting

Comments can be good in cases where the workflow is complex, but keep them useful, most developers can follow properly named steps and transitions.

Step & Transition Naming

Be thoughtful when creating steps and transitions, by naming a step that copies a file to a repository "copy file to repository" the code is self documenting. This also makes it easy when reading transitions to follow where the code is going next.

Versioning

Versioning can be tricky. Manual Versioning is best kept in the description area, where it is at the top of the workflow and easy to read. Better yet, use a versioning software like github to hold you code, comments, and utilize the PR system to code check if you have a team. This will also keep a nice history and changeling of what may have happened with your workflows over time.

log statements - execution labels

in short, use them. Fill out your execution labels, include helpful variables, this will make debugging the workflow much easier as you will know what certain values are at the step level.

'end' step no-op

it may seems silly to end your workflow with a step named 'end' but it is actually quite useful to know your workflow completed, and is a quick reference both to you and the user that this workflow completed.

Advanced Topics

With Great Power Comes Great Responsibility

Recently we introduced the ability to script inside workflow, allowing the use of Groovy to expand the ability of the Reach Engine Studio Platform. By reaching outside to a powerful language such as Groovy, we can perform tasks that would be quite difficult in the XML workflow language with ease inside Groovy. For Example, if I wanted to split up a filename into parts and structure that into a JSON to use the `saveDataObjectStep` it would be rather mind boggling to do it in our XML language. I could reach outside Workflow to a `RunCommand` step and pass some type of data back to the `resultDataDef`, but what if I happen to need more variables than just a simple `#baseFilename` passed to my script? This is where Groovy starts to make sense. Lets take a look at a relatively simple example:

The `groovyStep` allows me to access the variables in the workflow like `basename`, then I can run java and groovy inside the script, and when I'm done pass back to a `resultDataDef` that I can continue use inside the workflow. A different approach is to write the groovy script to a file, then call the file as part of the `groovyStep` definition, allowing us to repurpose this script in more than one workflow.

```
<groovyStep
  name="parse metadata"
  resultDataDef="jsonMetadata">
  <transition condition="true">
    <targetStepName>end</targetStepName>
  </transition>

  <script>
<![CDATA[
// split filename
def ba = basename.split("_");
def size = ba.size()
java.text.SimpleDateFormat sdf = new java.text.SimpleDateFormat('yyyyMMdd');
Calendar cal = Calendar.getInstance()

if(size == 9) {
  try {
    jsonMetadata.put("gameType", ba[0])
    if(ba[1].length() == 4){
      jsonMetadata.put("gameNumber", "${ba[0]}${ba[1]}")
    } else {
      jsonMetadata.put("gameNumber", "${ba[0]}0${ba[1]}")
    }
    jsonMetadata.put("awayTeam", ba[2])
    jsonMetadata.put("homeTeam", ba[3])
    jsonMetadata.put("season", ba[4])
    jsonMetadata.put("feedCode", ba[5])
    jsonMetadata.put("prodIdentifier", ba[6])
    jsonMetadata.put("bitrate", ba[7])
    jsonMetadata.put("aspect", ba[8])
  } catch(e) { println "Parse Error: ${e}"; }
}
println jsonMetadata;
return jsonMetadata;
]]>
</script>
</groovyStep>
```

Coding Outside of Workflow:

Overview

The workflow has a lot of functionality available through the well defined workflow steps; however, sometimes there is extra functionality that is required (or easier) to be used that is not available through normal workflow steps. The well defined workflow steps provide error handling, error messages, and safe interactions with reachengine data objects. When deciding to use outside resources, it can become very difficult to debug without providing your own error handling and error messages. It is also very unsafe to access Reach Engine data objects outside of Reach Engine workflow.

Best Practices

Below are recommendations on when to step outside of the workflow language, when to stay away from an outside language, and how to debug in an outside language.

When to step out of workflow code:

- Parsing
- Difficult looping

Stay away from:

- Using reachengine data objects
- Using an outside language for things workflow can do easily

Debugging Tips:

- Familiarize yourself with log4j and how reachengine uses log4j to output to the RE log.
- Integrate into the log4j settings used by the RE log to output needed debugging messages to the same place all other workflow logging outputs to.
- output many log4j "DEBUG" statements throughout the code to help in depth debugging.
- output the major tasks as log4j "INFO"
- output problems as log4j "WARN" and stoppers as log4j "ERROR"
- Use a try/catch method (or some similar method depending on the language) to catch errors more easily.

log4j references:

- [Logging Workflows](#)
- [Creating Custom Log Files](#)

Outside Languages

Groovy

Groovy is a popular choice because there is a `groovyStep` defined to allow the workflow writer to step outside of the workflow language. This step can easily take the return value of the groovy code and put that value into a context data def.

Other (Run Command)

Any language can be integrated into workflow using a `runCommandStep` that calls an executable file/script (including groovy if you feel it is better to have the groovy code in a script rather than the workflow `groovyStep`).

Workflow Creation:

Test Step — Hello World

Create a workflow that prints “Hello World” to the log and the execution label.

Ingest Directory

Ingest a directory of assets into Studio

Ingest Directory part 2

setup a cron to run your directory ingest

Set (Select) Metadata on existing Asset

Utilize a workflow to add metadata to a selected asset in the UI

Read a JSON file into a variable

Read in a JSON formatted file and print the contents out to the log.

Combine Read & Set

Read in a JSON formatted file and set the JSON metadata onto a selected asset

Export Timeline with user specified transcode setting

Create a pick list with transcode settings, present them to the user, and have the user select one , then export a timeline in the selected transcode format

Advanced — Query for the asset based on some property

Include the asset name in your JSON file, select the JSON file, find the asset, then set the metadata on that asset

Advanced — Set a piece of metadata through API

Use the httpStep to hit the reach engine API, authenticate, then set a piece of metadata on an asset.

page is intentionally left empty.